digital whisper
Technologic Papers

Trust no package: זיהוי נוזקות ב-NPM

מאת ליאורה רבייב

הקדמה

בעולם הפיתוח המודרני, אנחנו בונים תוכנה כמו לגו. פקודה אחת קטנה בטרמינל - npm install - ועשרות, אם לא מאות, חבילות צד-שלישי נשאבות לתוך הפרויקט שלנו. הנוחות הזו ממכרת, אבל היא גובה מחיר אבטחתי כבד. בפועל, התרגלנו להעניק אמון עיוור בקוד שאנחנו לא באמת מכירים, ולתת לו לרוץ עם הרשאות נרחבות על המכונה המקומית שלנו או בשרתי ה-CI/CD.

בשנה האחרונה, תוקפים הבינו את הפוטנציאל הטמון באמון הזה ושינו את חוקי המשחק של מתקפות שרשרת האספקה. הם כבר לא מסתפקים ב-Typosquatting פשוט. אנחנו רואים היום תולעים שמפיצות את עצמן אוטומטית דרך סקריפטים של postinstall (כמו תולעת "Shai-Hulud"), ומתקפות מתוחכמות שמתחזות לכלי AI פופולריים (כמו Claude Code) במטרה לגנוב Secret-ים ו-Token-ים של מפתחים.

המציאות הזו מחייבת אותנו לשנות תפיסה ולאמץ גישת Zero Trust גם כלפי מנהלי החבילות שלנו. התוקפים מסווים את הלוגיקה הזדונית שלהם היטב, משתמשים בטכניקות אובפוסקציה מורכבות, ומנצלים ספריות תמימות למראה כמסווה.

במאמר זה, אקח אתכם אל מאחורי הקלעים של פיתוח סורק איומים היוריסטי שכתבתי בפייתון, שמטרתו לצוד את החבילות הללו עוד לפני שהן מקבלות הזדמנות לרוץ. נצלול לארכיטקטורה של הסורק ונראה כיצד שילוב של ניתוח מטא-דאטה, זיהוי אנומליות בתלויות, ושימוש בנוסחה מתמטית לזיהוי של קוד מעורפל - יכולים לספק לנו קו הגנה ראשון ואפקטיבי מפני מתקפות NPM דומות.

מושגים בסיסיים

NPM - Node Package Manager

מנהל החבילות הרשמי והגדול ביותר של סביבת Node.js (ושל שפת JavaScript בכלל). הוא משמש כמאגר עצום המכיל מיליוני ספריות קוד פתוח מוכנות מראש - מכלים קטנים לחישוב תאריכים ועד למסגרות פיתוח ענקיות כמו React. כל מפתח יכול להעלות חבילה ל-NPM, וכל מפתח אחר יכול להוריד אותה.

Package.json

"תעודת הזהות" של כל חבילת NPM. זהו קובץ הגדרות המכיל מטא-דאטה על הפרויקט (שם, גרסה, מחבר), את רשימת החבילות האחרות שהוא תלוי בהן (Dependencies), ואת ה-Scripts - פקודות מערכת שניתן להריץ.

Install Scripts

מנגנון בתוך קובץ ה-package.json המאפשר לחבילה להריץ פקודות מערכת באופן אוטומטי. לדוגמה, סקריפט מסוג postinstall ירוץ אוטומטית מיד לאחר שהחבילה ירדה למחשב וסקריפט מסוג preinstall רץ רגע לפני שהחבילה מותקנת. במקור, המנגנון נועד לפעולות תשתית לגיטימיות, כמו קימפול ספריות C++ או הכנת סביבת העבודה. עם זאת, מנקודת מבט אבטחתית, זהו וקטור תקיפה אידיאלי: ברגע שהקורבן מקליד בטרמינל את הפקודה התמימה npm install, הסקריפט הזדוני מופעל אוטומטית באמצעות הרשאות המשתמש, עוד לפני שהמפתח הספיק לייבא אפילו שורת קוד אחת מהחבילה אל תוך הפרויקט שלו.

Dependency Tree

ב-NPM, חבילה א' יכולה להיות תלויה בחבילה ב', שתלויה בחבילה ג', וכן הלאה. התקנה של חבילה תמימה אחת יכולה "לשאוב" מאות חבילות אחרות אל תוך הפרויקט שלכם. תוקפים מנצלים את השרשרת הזו כדי להזריק קוד זדוני לחבילה קטנה ונידחת עמוק בעץ, מתוך ידיעה שהיא תגיע בסופו של דבר למשתמשי הקצה.

Axios

לאחרונה, בסוף מרץ 2026, ספריית Axios הפופולרית נפלה קורבן למתקפת שרשרת אספקה. קבוצת התקיפה הצפון-קוריאנית "Sapphire Sleet" הצליחה לשחרר גרסאות רשמיות ופגומות של החבילה (גרסאות 1.14.1 ו-0.30.4) מבלי לעורר חשד.

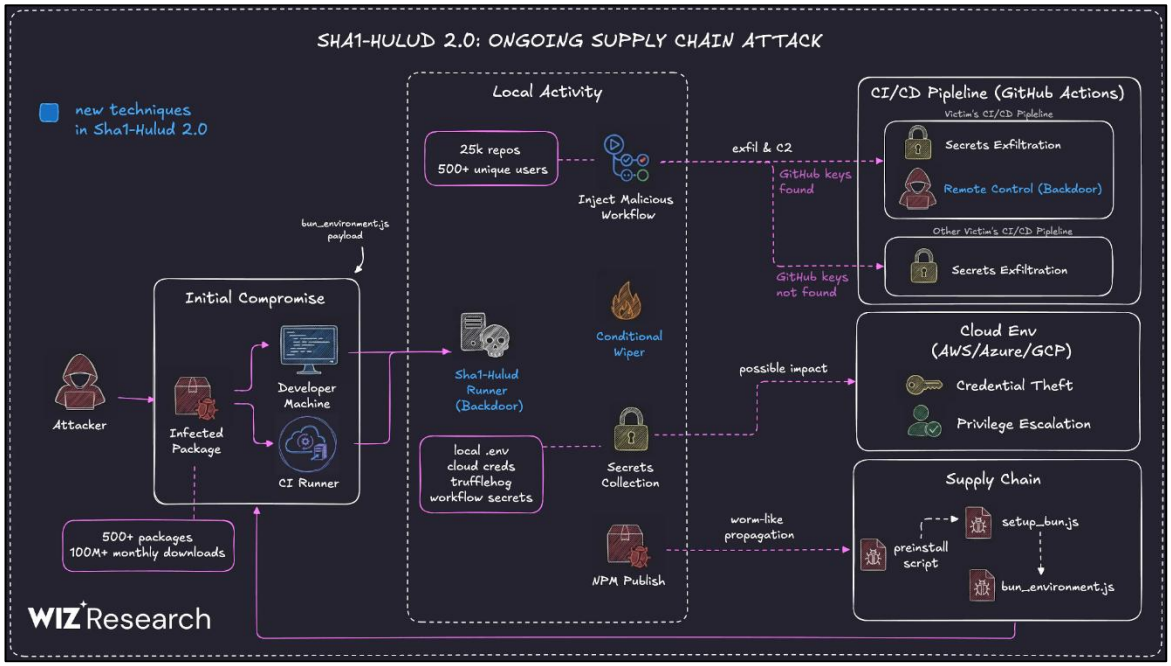
הייחודיות במתקפה זו הייתה שהתוקפים לא שינו אף שורת קוד בלוגיקה המקורית של Axios. במקום זאת, הם ביצעו הזרקת תלות (Dependency Insertion) והוסיפו ל-package.json תלות נסתרת וזדונית בשם plain-crypto-js. ברגע שמישהו התקין את העדכון של Axios, התלות הזדונית נמשכה אוטומטית והפעילה סקריפט בשם setup.js באמצעות מנגנון ה-post-install. הסקריפט פעל בשקט מאחורי הקלעים, הפעיל לוגיקה מעורפלת לזיהוי מערכת ההפעלה, והוריד סוס טרויאני (RAT) שאפשר לתוקפים שליטה מלאה מרחוק. אחד הדברים המדאיגים ביותר באירוע הזה הוא היכולת של הקוד למחוק את עקבותיו. לאחר שהווירוס חודר למערכת ומבצע את הקישור לשרת של התוקפים, הוא מוחק את קובץ ההתקנה המקורי ומשנה את שמות הקבצים בתיקיית הקוד שלכם כך שייראו תקינים לחלוטין.

גם אם ביקשתם מכלי AI לכתוב לכם אוטומציה או דף נחיתה והרצתם את הקוד אצלכם במחשב, יכול להיות שהכלי השתמש בגרסה החדשה ביותר של Axios. התקנה כזו לא דורשת אישור שלכם; היא קורית כחלק מהגדרות ברירת המחדל של סביבת הפיתוח.

מקרה זה ממחיש לנו בצורה מרתקת כיצד אפילו ספריות תשתית הנמצאות בלב ליבו של כמעט כל פרויקט מודרני יכולות להפוך בן רגע לווקטור תקיפה קטלני המחייב ניתוח מעמיק.

"Shai-Hulud" - תולעת שרשרת האספקה

במהלך המחצית השנייה של שנת 2025, המערכת האקולוגית של NPM חוותה שתי מתקפות שרשרת אספקה חסרות תקדים תחת השם "Shai-Hulud", שהדגימו לראשונה נוזקה הפועלת כתולעת (Worm) המפיצה את עצמה. בגל הראשון שהחל בספטמבר, התוקפים ניצלו את סקריפט ה-postinstall כדי להפעיל סורקי סודות פנימיים (כמו TruffleHog) שקצרו משתני סביבה, מפתחות ענן וטוקנים של GitHub ו-NPM. הסודות שנגנבו הועלו למאגרי GitHub פומביים שיצר התוקף, ובחלק מהמקרים התולעת אף הפכה מאגרים פרטיים לציבוריים (תחת הסיומת migration-). זמן קצר לאחר מכן, בנובמבר, הופיעה הגרסה ההרסנית יותר - "Shai-Hulud 2.0" - שהדביקה במהירות למעלה מ-25,000 מאגרים ופגעה בתלויות פופולריות של חברות ענק כמו Postman ו-Zapier. גרסה זו עברה לפעול כבר בשלב ה-preinstall (באמצעות קבצים כמו setup_bun.js), והציגה יכולות חסרות תקדים: היא ידעה להבדיל בין סביבת מפתח מקומית לסביבת CI/CD כדי להתחמק מניטור, ויצרה מנגנון Backdoor שאפשר לתוקפים להריץ פקודות מרחוק על המכונה הנגועה באמצעות פתיחת "דיונים" (Discussions) ב-GitHub. באופן מטריד, הגרסה השנייה גם ביצעה Cross-victim exfiltration, שבה סודות של ארגון אחד פורסמו בחשבון של קורבן אחר לגמרי. בשני הגלים, המטרה הייתה זהה: שימוש בטוקנים הנגנבים כדי לפנות ל-API של NPM ולפרסם אוטומטית גרסאות נגועות נוספות תחת שמם של מפתחים לגיטימיים, מה שיצר תגובת שרשרת אקספוננציאלית.



ארכיטקטורת הסורק

כדי להתמודד עם איומים כמו ניצול חבילה גדולה כמו Axios ו- Shai-Hulud , שרצים מיד עם ההתקנה (preinstall / postinstall) ומסווים את עצמם בעץ התלויות, בניתי סורק בפייתון המבוסס על ניתוח סטטי. הכלי סורק את קבצי המקור והמטא-דאטה ומחפש בהם התנהגויות אנומליות. מומלץ להריץ אותו על sandbox.

האתגר המרכזי בניתוח סטטי הוא הימנעות מ-False Positives. פונקציה כמו child_process.exec היא לגיטימית לחלוטין בכלי CLI, אך מחשידה מאוד בספרייה קטנה לחישוב תאריכים. לכן, הליבה של הסורק היא מערכת ניקוד משוקללת המחולקת לחמש קטגוריות סיכון מרכזיות. כל קטגוריה מקבלת ציון בנפרד (עם תקרת מקסימום כדי למנוע הטיה), ובסוף התהליך מחושב ציון כולל המכריע את רמת המסוכנות של החבילה.

חישוב הניקוד - ציון מבוסס אותות:

במקור, הסורק התבסס על חלוקה נוקשה של מכסות ניקוד (Caps) לכל קטגוריה, אך גישה זו יצרה "קופסה שחורה" שהקשתה להבין את חומרת האיום ולכן החלטתי לשנות אותה.

כדי לפתור זאת, שכתבתי את מנוע הסריקה כך שיעבוד בשיטה של **ציון מבוסס אותות (Signals)**. במקום להחזיר מספר אטום, המערכת אוספת "אותות סיכון" קונקרטיים במהלך הסריקה (למשל: "התגלו פקודות מסוכנות ב-Install hook", "נמצאו תלויות עודפות", או "התגלתה אובפוסקציה"). כל אות כזה תורם ניקוד ספציפי לציון הכולל, והחשוב מכל - מציג לאנליסט בסוף הריצה פירוט שקוף לחלוטין של **למה** החבילה סומנה כמסוכנת. גישה זו מייעלת את תהליך ה-Triage וחוסכת זמן תחקור יקר.

```python
def calculate_final_score(self) -> dict:
    """Signal-based scoring: add risk points, apply trust discount, clamp 0-100."""
    score = 0
    signals = []

    # 1. Install hook with dangerous commands
    if self.install_hook_score > 0:
        score += 30
        signals.append(("Install hook has dangerous commands", 30))

    # 2. Density of critical files
    if self.js_files_analyzed > 0 and self.files_with_critical > 0:
        density = self.files_with_critical / self.js_files_analyzed
        if density > 0.05:
            score += 5
            signals.append((f"File density > 5%", 5))

    # ... (Additional risk signals are calculated and appended here) ...

    subtotal = score

    # 3. Trust discount - The most important mechanism against False Positives
    if self.is_established:
        score = int(score * 0.6)
        signals.append((f"Established package discount (x0.6)", score - subtotal))

    # 4. Clamp final score between 0 and 100
    final = max(0, min(100, score))

    return {"score": final, "signals": signals, "subtotal": subtotal}

    def calculate_final_score(self) -> dict:
        """Signal-based scoring: add risk points, apply trust discount, clamp 0-100."""
        score = 0
        signals = []
```

אינטליגנציה היוריסטית: הבנת הקונטקסט

כדי לנסות להפוך את הסורק לחכם באמת, הוא אינו מסתמך רק על איסוף נקודות עיוור. הוספתי לוגיקה שמתאימה את רמת ההחמרה בהתאם לסוג החבילה. לדוגמה:

- **ספריות עזר:** אם הסורק מזהה שמדובר בספריית עזר (כמו lodash), הוא מצפה שהיא לא תכיל postinstall מורכב. במקרה כזה, עונש על סקריפטים או חוסר במטא-דאטה יהיה חמור פי 1.5. מצד שני, קוד של ספריות עזר לרוב עובר מניפיקציה (Minification), ולכן הסורק "יסלח" להן קצת יותר בסעיף האובפוסקציה כדי למנוע התראות שווא.
- **החרגות לסביבות פיתוח מוכרות:** חבילות בסיס של סביבות פיתוח ענקיות (כמו React, Vue, Angular או Lodash) נוטות להיות מורכבות ולעתים חסרות תלויות לחלוטין (Zero dependencies). הסורק מכיל מילון החרגות מובנה (FRAMEWORK_EXCEPTIONS) שיודע לזהות את החבילות הללו ולא להעניש אותן על מבנה עץ התלויות שלהן, ובכך חוסך מאיתנו התראות שווא מיותרות.

- **מדד הצפיפות (Density Score):** זיהוי איומים הוא לא רק מציאת מילה מחשידה, אלא הבנת היקף ההדבקה ביחס לגודל החבילה. מילה מסוכנת אחת מתוך 10,000 קבצים לגיטימיים יכולה להיות הערה תמימה בקוד או אזכור מקרי. לעומת זאת, אם 5% או 25% מהקבצים בחבילה מכילים קריאות קריטיות - מדובר ככל הנראה בהדבקה רוחבית מכוונת. הסורק מחשב את צפיפות הקבצים הנגועים ומוסיף "קנס" משמעותי לציון הסיכון ככל שהצפיפות עולה.
- **הנחת הנאמנות (Trust Discount):** זהו המנגנון החשוב ביותר להפחתת רעש ולמניעת התראות שווא בחבילות ענק (כמו React או Lodash). הסורק בוחן את המטא-דאטה של החבילה - אם יש לה גרסה בוגרת (מעל 1.0.0), רישיון תקין, תיעוד עשיר (מעל 30 תווים), וקישור לריפוזיטורי מקור (כמו GitHub) - הסורק מחיל עליה אוטומטית "הנחת נאמנות" של 40% (מכפיל את הציון הכולל ב-0.6). הרציונל הוא שחבילות מבוססות ופומביות כאלו נוטות לעבור בקרת קהילה מחמירה יותר, ולכן סבירות ההדבקה הראשונית (Typosquatting) שלהן נמוכה משמעותית מחבילות אנונימיות לחלוטין.

```
            # Check for new packages
            if "version" in data and data["version"] == "0.0.1":
                # New utility libraries are more suspicious
                multiplier = 2.0 if self.is_utility_library else 1.0
                self.results["metadata"] += int(10 * multiplier)
                logger.info("Note: Very new package (version 0.0.1)")
```

הפונקציה analyze_scripts

כפי שראינו במקרה של תולעת Shai-Hulud, וקטור התקיפה המרכזי בשרשרת האספקה של NPM נשען על הרצה אוטומטית בעזרת Lifecycle Scripts. כדי להתמודד עם האיום הזה של "Zero-Click", הסורק קורא את קובץ ה-package.json של החבילה ומנתח את שדה ה-scripts.

הפונקציה analyze_scripts מנתחת את התוכן של קוד ההתקנה בחיפוש אחר תבניות פעולה של Droppers (נוזקות שמורידות את השלב הבא של המתקפה מהרשת) ו-Reverse Shells.

כך נראית הפונקציה:

```python
def analyze_scripts(self):
    """Analyze package.json scripts with install hook focus"""
    package_json = self.package_path / "package.json"
    if not package_json.exists():
        logger.warning("package.json not found in the package directory")
        return

    try:
        with open(package_json) as f:
            data = json.load(f)
            scripts = data.get("scripts", {})
            if not scripts:
                logger.info("No scripts found in package.json")
                return

            for name, cmd in scripts.items():
                is_auto_run_hook = any(hook in name for hook in ["preinstall", "postinstall", "install"])

                for keyword, weight in KEYWORD_WEIGHTS.items():
                    if keyword in cmd:
                        hook_multiplier = 2.0 if is_auto_run_hook else 1.0
                        adjusted_weight = int(weight * hook_multiplier)
                        self.results["scripts"] += adjusted_weight
                        if is_auto_run_hook and weight >= 15:
                            self.install_hook_score += adjusted_weight
                        risk_level = "CRITICAL" if is_auto_run_hook else "HIGH"
                        logger.warning(f"[{risk_level} SCRIPT RISK] Found '{keyword}' in script: {name}")
                        if weight >= 20 and f"{keyword} in {name}" not in self.critical_findings:
                            self.critical_findings.append(f"{keyword} in {name}")
```

מה עשינו כאן?

1. **חיפוש אקטיבי של כלי רשת ו-Shells:** מילון המשקלים (KEYWORD_WEIGHTS) מכיל פקודות שלרוב אין להן שום הצדקה בספריית JavaScript סטנדרטית. קריאה ל-curl או wget מתוך סקריפט מצביעה על ניסיון משיכת קבצים זדוניים משרת C2. קריאה ל-nc מצביעה על ניסיון לפתוח חיבור ישיר לתוקף.
2. **משקל כפול על אוטומציה:** אם הפקודות החשודות האלו מופיעות בסקריפט שמופעל ידנית (כמו npm run test או npm run build), החבילה סופגת ניקוד שלילי, אך לא בהכרח קריטי. לעומת זאת, אם הסקריפט מוגדר כ-preinstall או postinstall (כלומר, הוא ירוץ אוטומטית ללא ידיעת המפתח), הלוגיקה מכפילה את העונש (hook_multiplier = 2.0). יתרה מזאת, המערכת רושמת את הממצא, ומערכת ה-Signals הסופית תזהה שמופעל "Install hook" מסוכן, מה שיקפיץ מיד את רמת הסיכון של החבילה לאדומה באמצעות קנס ישיר של 30 נקודות.

אובפוסקציה ו-JSFuck

הנחת העבודה שלנו בניתוח חבילות NPM היא שתוקף מקצועי לעולם לא ישאיר קוד זדוני גלוי לעין. פקודות כמו eval() או קריאות ל-child_process שיקפצו מיד בסריקה פשוטה, יעברו תהליך של הסוואה או אובפוסקציה. המטרה של התוקף היא להפוך את הקוד לקריא עבור מנוע ה-JavaScript (ה-V8), אך לבלתי קריא לחלוטין עבור חוקר אנושי או סורק מבוסס מילות מפתח.

כדי להתמודד עם זה, הסורק שפיתחתי משתמש בגישה דו-שלבית. השלב הראשון הוא חיפוש אקטיבי של תבניות אובפוסקציה מוכרות תוך שימוש בביטויים רגולריים (Regex).

במקום גישה בינארית של "מצאתי/לא מצאתי", הסורק עובד עם מערכת משקלים. כל תבנית שמזוהה מוסיפה נקודות לציון ה"ערפול" הכולל של הקובץ. אם הציון חוצה רף מסוים, הקובץ מסומן כחשוד ומוסיף משקל לציון הסיכון הכללי של החבילה.

הנה רשימת התבניות המרכזיות שהסורק מחפש, כפי שהן מוגדרות בקוד:

```
61  OBFUSCATION_PATTERNS = [
62      (r'eval\(function\(p,a,c,k,e,d\)', 25),  # Packer.js
63      (r'\\x[0-9a-fA-F]{2}', 15),              # Hex encoding
64      (r'\b[a-zA-Z0-9]{25,}\b', 10),           # Very long random identifiers
65      (r'\[\]\["\w+"\]\("\w+"\)', 25),         # JSFuck patterns
66      (r'String\.fromCharCode\(\d+(?:,\d+)+\)', 20),  # Character code arrays
67      (r'(?:\\u[0-9a-fA-F]{4}){5,}', 15),      # Multiple unicode escapes
68      (r'=~\[\]', 25),                          # JSFuck operators
69      (r'\(!""\+\[\]\)', 25),                   # JSFuck number obfuscation
70      (r'atob\([^)]{20,}\)', 20),              # Long base64 strings
71      (r'(?:["\'][^"\']{1,2}["\']\s*\+\s*){10,}', 15),  # Split strings
72      (r'\breturn\s*/[^/]+/\.exec\(', 20),      # Regex-based deobfuscation
73      (r'window\[(["\']\w+["\']\+["\'\w+]+)\]', 20),  # Dynamic property access
74  ]
```

הרשימה כמובן יכולה להיות גדולה יותר ועשירה יותר, אבל אלו התבניות שבחרתי כרגע להתמקד בהן.

מה אנחנו בעצם מחפשים כאן?

1. **קידודים חריגים (Hex/Unicode):** התבניות \\x[0-9a-fA-F]{2} ומרבה תווי ה-Unicode נועדו לתפוס קוד שמסתיר מחרוזות. תוקפים משתמשים לעיתים קרובות בקידודים אלו כדי להסתיר כתובות IP, דומיינים (C2) או פקודות מערכת.
2. **JSFuck:** טכניקת tucpuxemhv קיצונית ומרתקת שנקראת JSFuck מאפשרת לכתוב כל תוכנית JavaScript חוקית באמצעות 6 תווים בלבד: + ! ( ) [ ]. תוקפים משתמשים בה כדי לעקוף סורקים שמחפשים מילים באנגלית. התבניות \[\]\+\!"" (המייצגת יצירת מספרים) ו-\("\w+"\)\[\]\["\w+"\] מצליחות לצוד את הניסיונות האלו ולהעניק להם משקל קריטי של 25 נקודות.
3. **מחרוזות מפוצלות (Split Strings) וקריאה דינמית:** כדי לעקוף חיפוש פשוט של המילה require או exec, תוקפים יפצלו את המחרוזת ויחברו אותה בזמן ריצה (למשל: 'ex' + 'ec'). השורה של ה- Split strings וקריאת המאפיינים הדינמית (window[...]) נועדו לזהות קוד שעסוק מדי בחיבור מחרוזות קטנות בצורה חשודה.
4. **משתנים אקראיים (Long random identifiers):** כלים שמבצעים ערפול קוד נוטים להחליף שמות פונקציות ומשתנים לשמות ארוכים וחסרי משמעות למשל (aBcDeFgHiJkL...) . הסורק מזהה מילים מעל 25 תווים רצופים כאינדיקטור (במשקל נמוך יותר של 10 נקודות, מחשש ל-False Positives בקוד שמקורו ב-WebAssembly או קבצים דחוסים מאוד).

כדי לשפר ביצועים, כל הביטויים הרגולריים האלו מקומפלים (Compiled) מראש בתחילת הריצה:

```
# Compile regex patterns once for efficiency
COMPILED_OBFUSCATION_PATTERNS = [(re.compile(pattern), weight) for pattern, weight in OBFUSCATION_PATTERNS]
```

אבל Regex, חזק ככל שיהיה, מוגבל בסופו של דבר למה שאנחנו יודעים לחפש. תוקף שיכתוב אלגוריתם ערפול חדש, עלול לחמוק מהתבניות שלנו. בדיוק בשביל זה, השלב השני של ניתוח הערפול מסתמך על מתמטיקה טהורה.

אנטרופיית שאנון (Shannon Entropy)

בפרק הקודם ראינו שניתן לצוד קוד Obfuscated באמצעות תבניות Regex מוכרות. הבעיה בגישה זו היא שאנחנו יכולים למצוא רק את מה שאנחנו כבר יודעים לחפש.

כדי להתמודד עם האזור העיוור הזה, הסורק מפעיל שכבת הגנה שנייה המבוססת על אנטרופיית שאנון- מושג מתורת האינפורמציה המודד את רמת האקראיות, או חוסר הוודאות, בתוך אוסף של נתונים. טקסט רגיל (וקוד אנושי) מכיל תבניות שחוזרות על עצמן (רווחים, מילים שמורות כמו function, return, const, ואותיות באנגלית שמופיעות בתדירות ידועה), ולכן האנטרופיה שלו נמוכה יחסית, לרוב סביב 3.5 עד 4.5. לעומת זאת, קוד שעבר הצפנה חזקה, קידוד Base64 מורכב, או אובפוסקציה קיצונית שדוחסת נתונים- יראה כמו בלילה אקראית לחלוטין של תווים. במצב כזה, מדד האנטרופיה יזנק למעלה.

הנוסחה:

$$H(x) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

final entropy | probability of an event | logarithm of event's probability

כך נראה המימוש של הנוסחה המתמטית בסורק:

```python
def calculate_entropy(self, text: str) -> float:
    if not text or len(text) < 100:  # Skip very short texts
        return 0

    # Use frequency table for faster calculation
    freq = {}
    for char in text:
        freq[char] = freq.get(char, 0) + 1

    length = len(text)
    entropy = 0
    for count in freq.values():
        p_x = count / length
        entropy += -p_x * math.log2(p_x)

    return entropy
```

איך זה עובד בפועל?

הפונקציה מבצעת מעבר יחיד על הטקסט ובונה טבלת שכיחויות של התווים (freq). לאחר מכן, היא מחשבת את ההסתברות ($P_$X) להופעת כל תו ביחס לאורך הכולל של הקובץ, וסוכמת את הערכים לפי הנוסחה: $H = - \sum p_x \log_2 p_x$. ככל שהפיזור אחיד יותר (יותר תווים שונים שמופיעים בתדירות דומה), כך התוצאה הסופית תתקרב לערך המקסימלי (אנטרופיה של 8 עבור בית בודד).

כדי למנוע False Positives על משתנים קצרים או קוד זעיר, הפונקציה מוגדרת לדלג על טקסטים הקצרים מ-100 תווים.

שילוב האנטרופיה בתהליך קבלת ההחלטות:

הציון שהתקבל מפונקציית האנטרופיה לא עומד בפני עצמו, אלא מוזן אל תוך הלוגיקה של detect_obfuscation. כאן קבענו Thresholds שמרניים יחסית, כדי להתמודד עם קוד שהוא Minified שלעיתים מציג אנטרופיה מעט גבוהה מהרגיל:

```
        # Calculate entropy for longer files
        if content_length > 500:
            entropy = self.calculate_entropy(content)
            if entropy > 5.5:  # Increased threshold for more precision
                obfuscation_score += 25
            elif entropy > 5.0:
                obfuscation_score += 15
```

ברגע שהקובץ חוצה רף אנטרופיה של 5.5 (רמה חריגה מאוד לקוד קריא), הוא סופג מיד קנס כבד של 25 נקודות למדד אובפוסקציה. השילוב של אנטרופיה יחד עם Regex מאפשר לסורק לצוד נוזקות מוסוות ברמת דיוק גבוהה בהרבה מסריקת טקסט פשוטה.

זיהוי אובפוסקציה רק על בסיס מתמטיקה עלול להוביל לסימון שגוי של קוד לגיטימי שעבר דחיסה (Minification). כדי למנוע זאת, הוספתי בפונקציה detect_obfuscation רשימת "סממני קוד נורמלי" (NORMAL_CODE_INDICATORS). הסורק מחפש מילות מפתח שגרתיות של מפתחים (כמו function, module.exports, reduce, forEach). אם הוא מזהה כמות מספקת של סממנים אלו בקובץ שאינו עצום בגודלו, הוא יניח שמדובר בקוד לגיטימי וידלג על ענישת האובפוסקציה.

```
def detect_obfuscation(self, content: str, file_path: str) -> bool:
    """Enhanced obfuscation detection with weighted scoring"""
    if self.should_ignore_file(file_path):
        return False

    # Skip files that clearly contain normal code (False Positive prevention)
    normal_indicators_count = sum(1 for indicator in NORMAL_CODE_INDICATORS if indicator in content)
    if normal_indicators_count >= 3 and len(content) < 10000:
        return False

    obfuscation_score = 0
    content_length = len(content)

    # ... (Proceed to Entropy calculation and Regex patterns) ...
```

ביצועים בעולם האמיתי

כשאנחנו מנתחים חבילת NPM, אנחנו לא מתמודדים עם קובץ אחד. Dependency Tree של פרויקט ממוצע, כמו אפליקציית React סטנדרטית, יכול להכיל עשרות אלפי קבצים בתוך תיקיית ה-node_modules.

אם הסורק יעבור על הקבצים הללו בגישה הטורית המסורתית (קריאת קובץ, הרצת Regex, חישוב אנטרופיה, ורק אז מעבר לקובץ הבא) - סריקת חבילה אחת עשויה לקחת דקות ארוכות, והיא תהפוך לצוואר בקבוק משמעותי בתהליך הפיתוח (Pipeline).

לכן, הארכיטקטורה של הסורק חייבת להיות **מקבילית (Concurrent).**

כדי להתמודד עם היקף הקבצים, הסורק עושה שימוש במודול concurrent.futures המובנה בפייתון, וליתר דיוק ב-ThreadPoolExecutor. גישה זו מאפשרת לנו לנצל מספר Threads הפועלים במקביל, ולקצר משמעותית את זמן הריצה הכולל:

```
def analyze_files(self):
    """Multi-threaded file analysis"""
    js_files = self.collect_js_files()
    self.js_files_analyzed = len(js_files)

    if not js_files:
        logger.info("No JavaScript/TypeScript files found in package")
        return

    logger.info(f"Analyzing {len(js_files)} JavaScript/TypeScript files...")

    file_score = 0
    obfuscation_count = 0

    # Use ThreadPoolExecutor for parallel processing
    with ThreadPoolExecutor(max_workers=self.max_workers) as executor:
        results = list(executor.map(self.analyze_file, js_files))

        for i, result in enumerate(results):
            file_score += result["score"]

            if result["obfuscated"]:
                obfuscation_count += 1

            # Log critical and high-risk findings
            for keyword, risk in result["keywords"]:
                if risk == "CRITICAL":
                    rel_path = js_files[i].relative_to(self.package_path)
                    logger.warning(f"[CRITICAL RISK] Found '{keyword}' in {rel_path}")
                    if keyword not in self.critical_findings:
                        self.critical_findings.append(keyword)
```

סכנת ה-Regex והגנת Timeouts:

מקביליות לבדה אינה מספיקה. בעולם הניתוח הסטטי ישנה סכנה מוכרת שנקראת **Catastrophic Backtracking** (נסיגה קטסטרופלית). מצב זה מתרחש כאשר Regex מורכב "מסתבך" בניסיון לנתח מחרוזת ארוכה שאינה תואמת לו לחלוטין. במקרים של קבצי JS ממוזערים שיכולים להכיל עשרות אלפי תווים בשורה אחת, ה-Regex עלול להיכנס ללולאת ניסיונות שעלולה לתקוע את הסורק (ואת ה-CPU) במשך שעות.

כדי למנוע מקובץ בודד לשתק את כל תהליך הסריקה, הסורק מיישם מנגנון **Timeout פנימי** (30 שניות לקובץ) עבור כל Thread:

```
    def analyze_file(self, file_path: Path) -> Dict:
        """Analyze a single JavaScript/TypeScript file with timeout protection"""
        try:
            with ThreadPoolExecutor(max_workers=1) as executor:
                future = executor.submit(self._analyze_file_internal, file_path)
                return future.result(timeout=FILE_ANALYSIS_TIMEOUT)
        except TimeoutError:
            logger.warning(f"Analysis timeout for file: {file_path}")
            return {"score": 0, "keywords": [], "obfuscated": False}
        except Exception as e:
            logger.error(f"Error analyzing file {file_path}: {str(e)}")
            return {"score": 0, "keywords": [], "obfuscated": False}
```

שילוב זה של עבודה מקבילית (max_workers) לצד הגבלת זמן קשיחה (timeout) מבטיח שהסורק יישאר דטרמיניסטי, מהיר ואמין, גם כשהוא נתקל בקוד סבוך במיוחד או בחבילות ענק, ומונע מתוקפים לנצל Regex Bombs כדי להשבית את מערך ההגנה.

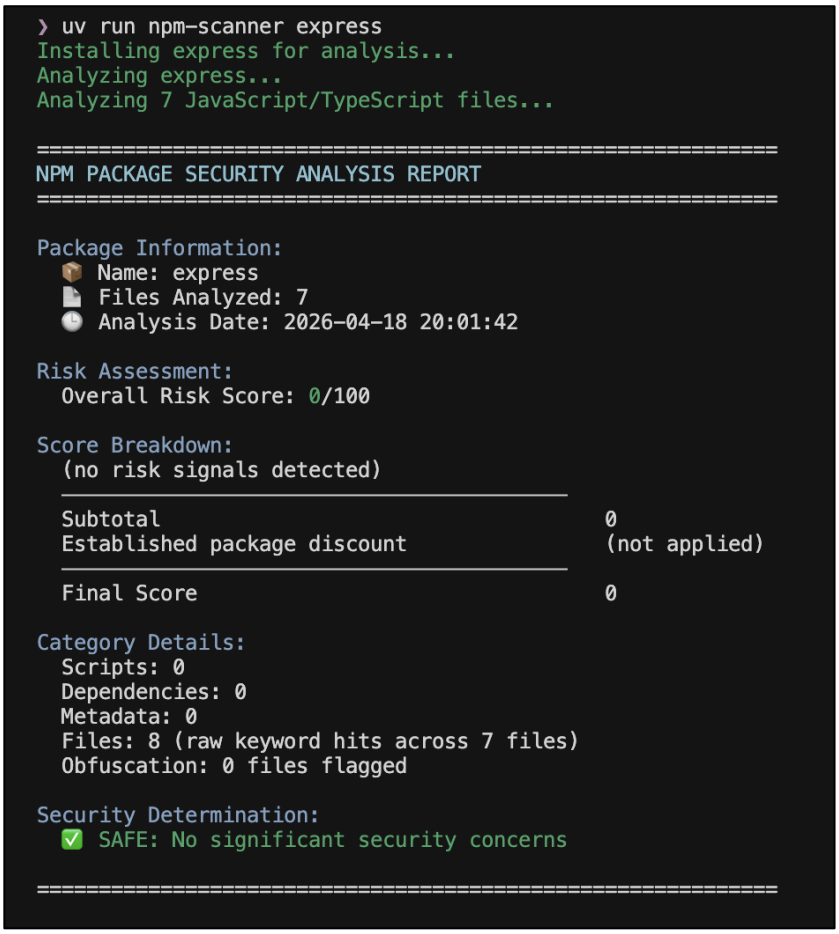
מקרי בוחן: הסורק בפעולה

אז עברנו על מבנה הסורק לעומק ואיך הוא עובד, למדנו כמה מושגים טכניים וקיבלנו מושג על התהליך. אבל מהן התוצאות בשטח?

החלטתי להדגים את יעילות הסורק על מספר חבילות מוכרות, אך מכיוון שחבילות זדוניות יורדות ישר מהמאגר של npm, יצרתי חבילות דמה שמתפקדות כחבילות חשודות.

1. express

express היא אחת ממסגרות הפיתוח הפופולריות ביותר ב-Node.js. כפי שניתן לראות, הסורק ניתח את הקבצים והציון **100/100**.



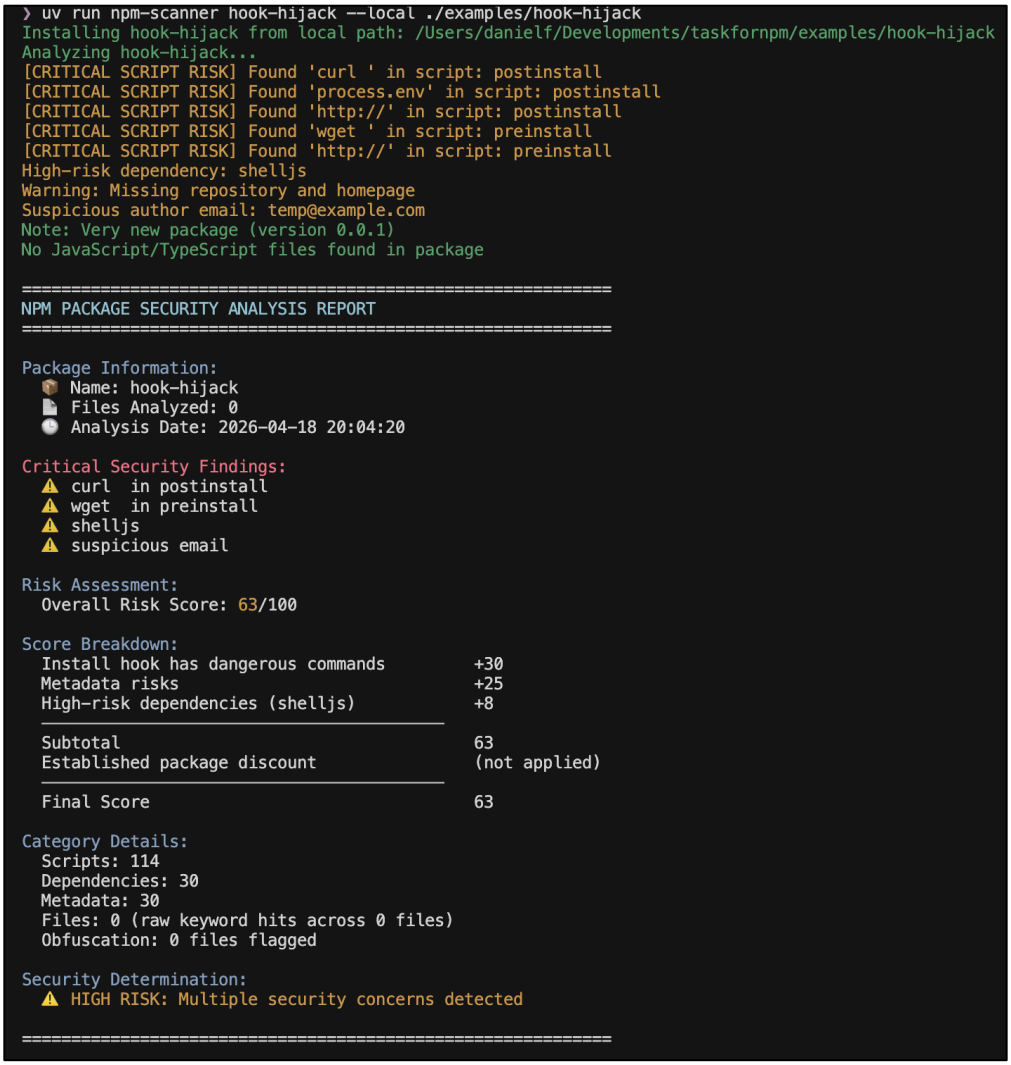
2. lodash

הסורק עבר על 1,048 קבצים וזיהה ממצאים שלרוב נחשבים כמחשידים: קריאות ל-fs.writeFile ושני קבצים שסומנו כ-Obfuscated. עם זאת, בזכות מדד הצפיפות החכם המערכת הבינה את המשקל המתאים לכך. הציון נשאר בתחום התחתון של הסקאלה: **23/100**.

```
❯ uv run npm-scanner lodash
Installing lodash for analysis...
Analyzing lodash...
[HIGH SCRIPT RISK] Found 'https://' in script: test
Note: Zero dependencies - generally good practice
Analyzing 1048 JavaScript/TypeScript files...
[CRITICAL RISK] Found 'fs.writeFile' in lodash.js
[CRITICAL RISK] Found 'fs.writeFile' in template.js

============================================================
NPM PACKAGE SECURITY ANALYSIS REPORT
============================================================

Package Information:
  📦 Name: lodash
  📄 Files Analyzed: 1048
  🕒 Analysis Date: 2026-04-18 20:00:34

Critical Security Findings:
  ⚠️ fs.writeFile

Risk Assessment:
  Overall Risk Score: 23/100

Score Breakdown:
  Critical keywords in files (1 unique)       +8
  Minor risk patterns in files                +5
  Obfuscated files detected (2)               +10
  ──────────────────────────────────────────────
  Subtotal                                    23
  Established package discount                (not applied)
  ──────────────────────────────────────────────
  Final Score                                 23

Category Details:
  Scripts: 1
  Dependencies: 0
  Metadata: 0
  Files: 115 (raw keyword hits across 1048 files)
  Obfuscation: 2 files flagged

Security Determination:
  ℹ️ LOW RISK: Minor concerns detected

============================================================
```

3. hook-hijack

חבילת הדמה הזו עושה שימוש לרעה במנגנוני ה-Lifecycle של NPM. הסורק זיהה מיד קריאות לכלים כמו curl ו-wget בתוך סקריפטים מסוג preinstall ו-postinstall. הפעולה הזו הפעילה את אות הסיכון החמור ביותר במערכת (Install hook has dangerous commands), שהוסיף אוטומטית 30 נקודות. יחד עם פגיעויות מטא-דאטה (כמו אימייל פיקטיבי), החבילה הוקפצה לציון **63/100**.

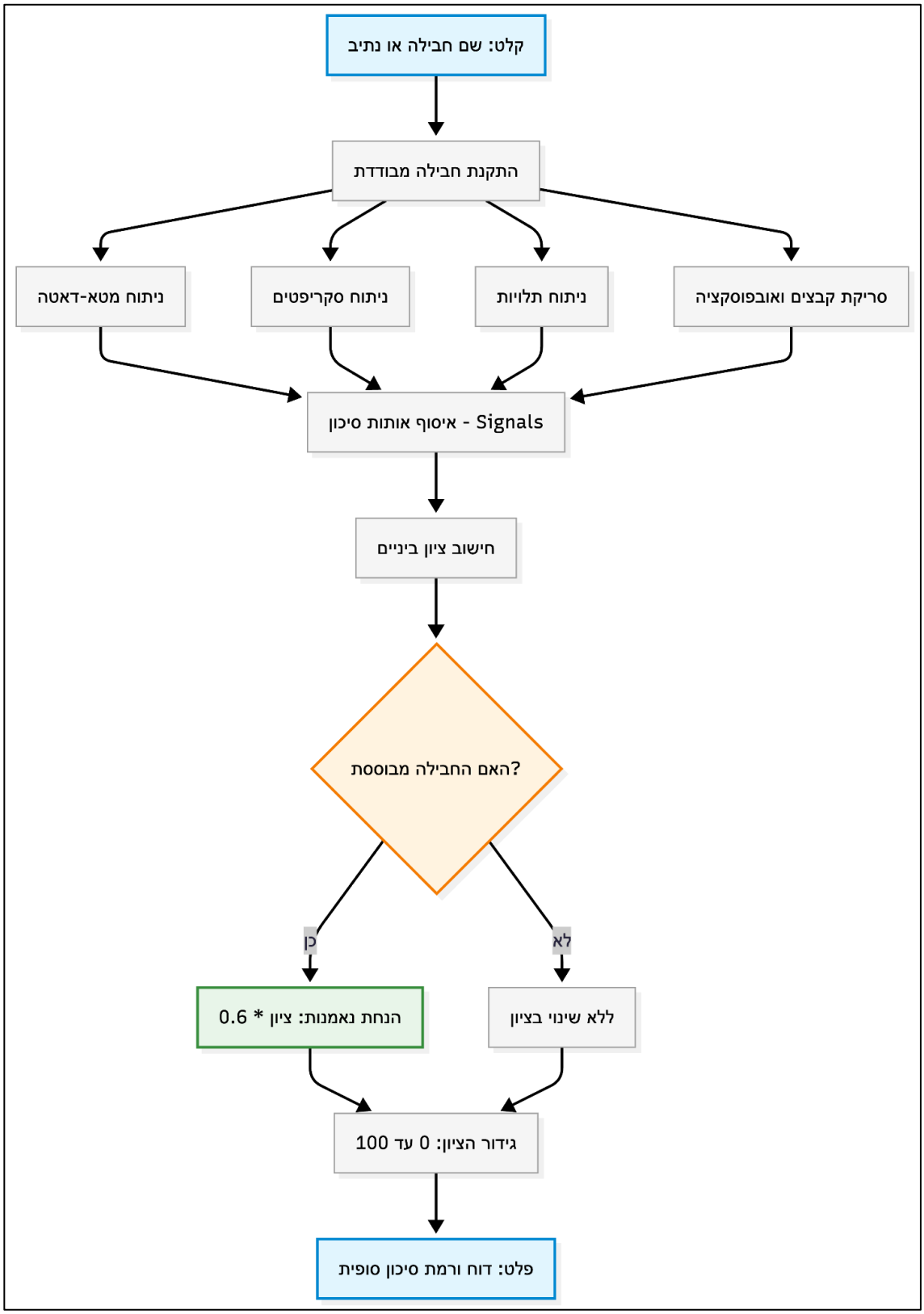


4. combined-attack

החבילת דמה מתארת חבילה מוכרת שנפרצה. מצד אחד, המערכת זיהתה שמדובר בחבילה מבוססת והעניקה לה "הנחת נאמנות" שהורידה 36 נקודות מהציון (Established package discount). מצד שני, כמות ה"רעש" הזדוני שהוזרק לחבילה הייתה עצומה: צפיפות קבצים נגועים גבוהה (50%), אובפוסקציה, ופקודות הרצה קריטיות (eval, child_process.exec). הציון הגולמי טיפס ל-136, כך שגם לאחר ההנחה, המערכת חתכה את הציון למקסימום האפשרי: **100/100.**

```
❯ uv run npm-scanner combined-attack --local ./examples/combined-attack

Installing combined-attack from local path: /Users/danielf/Developments/taskfornpm/examples/combined-attack
Analyzing combined-attack...
[CRITICAL SCRIPT RISK] Found 'curl ' in script: postinstall
[CRITICAL SCRIPT RISK] Found 'process.env' in script: postinstall
[CRITICAL SCRIPT RISK] Found 'http://' in script: postinstall
[CRITICAL SCRIPT RISK] Found 'wget ' in script: preinstall
[CRITICAL SCRIPT RISK] Found 'http://' in script: preinstall
High-risk dependency: shelljs
Warning: Missing repository and homepage
Suspicious author email: temp@example.com
Note: Very new package (version 0.0.1)
Analyzing 4 JavaScript/TypeScript files...
[CRITICAL RISK] Found 'eval(' in index.js
[CRITICAL RISK] Found 'child_process.exec' in index.js
[CRITICAL RISK] Found 'fs.writeFile' in index.js
[CRITICAL RISK] Found 'wget ' in index.js
[CRITICAL RISK] Found 'curl ' in index.js
[CRITICAL RISK] Found 'nc -' in index.js
[CRITICAL RISK] Found 'eval(' in lib/payload.js

============================================================
NPM PACKAGE SECURITY ANALYSIS REPORT
============================================================

Package Information:
  📦 Name: combined-attack
  📄 Files Analyzed: 4
  🕒 Analysis Date: 2026-04-18 20:04:50

Critical Security Findings:
  ⚠️ curl  in postinstall
  ⚠️ wget  in preinstall
  ⚠️ shelljs
  ⚠️ suspicious email
  ⚠️ eval(
  ⚠️ child_process.exec
  ⚠️ fs.writeFile
  ⚠️ wget
  ⚠️ curl
  ⚠️ nc -

Risk Assessment:
  Overall Risk Score: 100/100

Score Breakdown:
  Critical keywords in files (6 unique)       +48
  File density > 5% (2/4 = 50.0%)             +5
  File density > 25%                          +5
  Minor risk patterns in files                +5
  Install hook has dangerous commands         +30
  Obfuscated files detected (3)               +10
  Metadata risks                              +25
  High-risk dependencies (shelljs)            +8
  ──────────────────────────────────────────
  Subtotal                                    136
  Established package discount (x0.6)         -36
  ──────────────────────────────────────────
  Final Score                                 100

Category Details:
  Scripts: 114
  Dependencies: 30
  Metadata: 30
  Files: 188 (raw keyword hits across 4 files)
  Obfuscation: 3 files flagged

Security Determination:
  ⛔ CRITICAL RISK: Package exhibits highly suspicious behavior
```

תרשים ארכיטקטורת הסורק



מגבלות וכיווני פיתוח עתידיים

הסורק שהוצג במאמר זה אינו מתיימר להיות פתרון מושלם החסין בפני כל מתקפת שרשרת אספקה. משחק החתול והעכבר מול התוקפים ב-NPM הוא דינמי, ושיטות אובפוסקציה וההתחמקות רק ילכו וישתכללו. הכלי הזה נבנה כ-PoC מבוסס ניתוח סטטי, נועד לשמש כנקודת זינוק מחשבתית וטכנית. ישנם מספר כיווני פיתוח שניתן ורצוי להוסיף לסורק כדי להפוך אותו לחסין ומדויק יותר:

מעבר מניתוח טקסטואלי לניתוח סמנטי (AST): התבססות על Regex היא מהירה, אך מוגבלת לזיהוי תבניות טקסט. תוקף מתוחכם יכול לשכתב את הקוד כך שיחמוק מהתבניות שלנו. הצעד הבא הוא שילוב ניתוח עץ תחביר מופשט (Abstract Syntax Tree), שיאפשר לסורק להבין את המשמעות הלוגית מאחורי הפקודות ולא רק את המבנה הטקסטואלי שלהן.

אימות מטא-דאטה מול ה-API של NPM: כרגע, מנגנון ההיוריסטיקה מעניק אמון רב לקובץ ה-package.json המקומי. המשמעות היא שתוקף יכול לזייף חבילה איכותית על ידי הענקת גרסה גבוהה, הוספת רישיון מזויף, או שימוש בשם תמים המרמז על ספריית עזר. כדי לחשוף זיופים כאלו, נדרשת אינטגרציה מול מאגרי NPM כדי להצליב נתוני טלמטריה חיים - כגון קצב הורדות בפועל, תאריך פרסום מקורי ודירוג המפתח.

בדיקות התנהגותיות (Dynamic Analysis): ניתוח סטטי עיוור למה שקורה בזמן ריצה. עטיפת תהליך הסריקה ב-Sandbox מבודד תאפשר לנו לחקור את החבילה בסביבה מבוקרת. כך נוכל לנטר בזמן אמת ניסיונות התחברות לשרתי C2 חיצונים או קריאה של משתני סביבה רגישים.

למידת מכונה (ML): אימון מודל AI על Datasets של רבבות חבילות לגיטימיות מול חבילות שסומנו כזדוניות, יאפשר למערכת ללמוד לזהות אנומליות בצורה אוטומטית ודינמית.

שילוב מודלי שפה (LLM as a Judge): סריקה סטטית של עשרות אלפי קבצי node_modules באמצעות API של מודל שפה אינה פרקטית מבחינת עלויות וזמני ריצה (Latency). עם זאת, הסורק ההיוריסטי המהיר שלנו ישמש כ"מסנן ראשוני" (1 Tier), ורק קבצים בודדים שיחצו את רף האנטרופיה או יסומנו כחשודים, יישלחו לניתוח מעמיק ב-LLM לצורך ביצוע Deobfuscation אוטומטי והבנת כוונת התוקף.

סיכום

מתקפות כמו Shai-Hulud והופעתן של תולעי שרשרת אספקה הוכיחו שהאמון העיוור בפקודה התמימה לכאורה npm install הוא מסוכן. תוקפים לא צריכים יותר לפרוץ לשרתי הארגון - מספיק להם לשתול קוד מעורפל בתוך סקריפט של חבילה נידחת, והפלטפורמה עצמה תעשה עבורם את עבודת ההפצה וההדבקה.

הסורק שפיתחנו במאמר זה מדגים כיצד ניתן באמצעות ניתוח סטטי מבוסס פייתון, אלגוריתמיקה, וחלוקת ניקוד חכמה - להאיר את הפינות החשוכות של תיקיית ה-node_modules. מנגנוני ביצועים כמו סריקה מקבילית ו-Timeouts הופכים כלים כאלו לפרקטיים ומהירים אפילו בסביבות CI/CD כבדות.

למרות שפתרונות ניתוח סטטי אינם חסינים לחלוטין ממעקפים, הם מהווים כיום קו הגנה הכרחי של שפיות ובקרה מול אקוסיסטם קוד פתוח שהפך לשדה מוקשים. הכלי הזה נבנה כ-PoC שנועד לשמש כנקודת זינוק מחשבתית וטכנית. המטרה היא להחזיר את השליטה לידיים שלנו, ואני מזמינה את הקהילה לקחת את הקונספטים הללו, לאתגר אותם, ולהמשיך לפתח את הדור הבא של כלי ההגנה.

מקורות מידע

- https://www.wiz.io/blog/shai-hulud-2-0-ongoing-supply-chain-attack
- https://www.wiz.io/blog/shai-hulud-npm-supply-chain-attack
- https://he.wikipedia.org/wiki/%D7%90%D7%A0%D7%98%D7%A8%D7%95%D7%A4%D7%99%D7%94_(%D7%A1%D7%98%D7%98%D7%99%D7%A1%D7%98%D7%99%D7%A7%D7%94)
- https://jsfuck.com/
- https://safedep.io/npm-sandworm-mode-supply-chain-attack/
- https://unit42.paloaltonetworks.com/npm-supply-chain-attack/
- https://www.microsoft.com/en-us/security/blog/2026/04/01/mitigating-the-axios-npm-supply-chain-compromise/
- https://www.huntress.com/blog/supply-chain-compromise-axios-npm-package